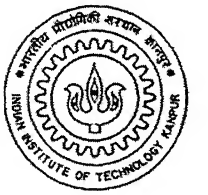
# Testing Changes made to the Code using Coverage Data

*A Thesis Submitted*
*in Partial Fulfillment of the Requirements*
*for the Degree of*
*Master of Technology*

*by*

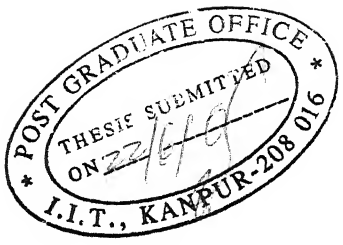
**Raghu Lingampally**




*to the*

**Department of Computer Science & Engineering**
Indian Institute of Technology, Kanpur

**June, 2005**




# Certificate

This is to certify that the work contained in the thesis entitled "*Testing Changes Made To The Code Using Coverage Data*", by *Raghu Lingampally*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

P. Jalote

June, 2005

(Dr. Pankaj Jalote)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

## Abstract

Testing is a mandatory process in software life cycle. Coverage analysis is one of the important processes during testing. There are several possible coverage measures that are existing. In this thesis, we have developed a coverage analysis tool that gathers the information on class, method, block, branch and predicate coverages using bytecode instrumentation. The information is maintained for individual test cases as well as for the entire test suite. Our implementation is done at bytecode level making it more efficient than other existing tools.

During testing, bugs are detected and corrected. The code often gets modified and it needs re-testing. Hence a new test suite has to be built for the changed code. The test suite which has already been used for the older version contains important artifacts which can be reused in the newer version of the program. Identifying the useful test cases for testing the modified code is very important aspect, which can potentially avoid re-building of test suite. Using the developed coverage analyzer, we propose an efficient test case prioritization scheme. The execution of program is tracked down with coverage data, using which a better and optimal test suite is built with test case prioritization.

Our experiments with a few programs suggest that the coverage value increases as logarithmic value of number of test cases. Also, changed code can be effectively tested by applying the proposed prioritization scheme on the already built test suite.

# Acknowledgments

I take this opportunity to express my sincere gratitude towards my thesis supervisor Dr. Pankaj Jalote for his invaluable guidance. It would have never been possible for me to take this project to completion without his innovative ideas and encouragement.

I also wish to thank whole heartily all the faculty members of the Department of Computer Science and Engineering, IIT Kanpur for enhancing my knowledge. I would like to thank whole of the mtech2003 batch for the times I shared with them.

My special thanks to Chaitanya and Aditya Varma for helping me whenever I faced problems in developing the tool. I would like to thank Atul Gupta for encouraging me to do many things. I would like to thank everyone worked in the CSE lab for providing a nice and challenging work environment.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Software testing is the process of identifying the conformance of the actual behavior of the software with the predicted and expected behavior. By doing so, we can identify and remove the defects so as to increase the correctness and make it more reliable.

To perform testing, we need a set of test cases, together called as test suite, which is used to evaluate the software. A test case consists of a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. The success of testing in revealing the defects and thus fixing them depends on the selection of the test cases. The test suite has to be designed in such a way that it can capture the properties of the software to the maximum extent.

There are two basic approaches of testing: Black-box testing and White-box testing. Black box testing does not consider the internal details of the software modules. It is restricted to verify the properties of a module as a whole. The specifications and requirements of a module are used to design the test case. Hence it will not involve the internals of the module[13].

The motive of white box testing is to exercise the implementation of the program. In white box testing, also called as structural testing, we try to test the data structures and program constructs. Common white box testing strategies include control flow based and mutation testing. Whitebox testing is the focus of this thesis,

whitebox testing for JAVA programs in particular.

## 1.1 Coverage Analysis

For a program, various coverage measures can be calculated and there are diverse set of tools for doing the same. The coverage measures can be broadly classified into control flow and data flow coverage.Control flow coverage can be further classified into:

***Package coverage***: Indicates whether at least one class in the package is covered or used.

***Class coverage***: This measure indicates if an object of a class is used in any of the test cases.

***Method coverage***: Method coverage indicates whether a method is invoked at least once. It allows quick identification of untested code.

***Block and statement coverage***: This measure indicates whether a set of contiguous program statements, called as blocks, are executed in the testing. (Each block is treated as atomic unit). In statement coverage the granularity is drilled down to statement level.

***Branch coverage***: It checks if all the branches in the program are executed or not. This checks if all the branches of a control flow statement are executed for some test data. Control flow statements can be: If ; If-else ; gotos; loops; switches; call invokes; return; break; Throw and catch exceptions.

**Coverage measures based on predicates**: Predicates are the conditions in the program which decide the execution path for particular test case input. These predicates classify the testing input domain into different categories. Based on predicates coverage measures can be categorized as follows

- *Relational operator coverage*:This coverage measure checks if every relation operator is executed.
- *Condition coverage*: Condition coverage reports the outcome of every boolean sub-expression each of which may be separated by some logical operator. To do this, each sub expression is measured independently.

- *Decision coverage*: This measure reports if the boolean expression tested in control flow statement is evaluated to both true and false values. Here the entire boolean expression is treated as a single entity irrespective if it's constituting sub expressions.

- *MCDC*: The Modified Condition Decision Coverage (MCDC) reports coverage of occurrences in which triggering atomic condition should also result in change of out come from true to false or false to true.

- *Predicate coverage*: It measures coverage of all possible combinations of every boolean expression. A boolean expression may contain embedded logic.

***Path Coverage***: This coverage measure checks to see if all the feasible paths in the program are taken in any of the test cases.

Data flow coverages are measured based on the usage of variables in the code. In general, an executable statement where a variable is assigned a value is called its definition. The following are some widely used coverage measures based on data flow.

***Data Definition-use Coverage***: The percentage of data definition-use pairs in a component that are exercised by a test case suite.

***Data Definition-use Pair***: This measure checks if a data definition is being used at some point during execution.

***C-use (computational) coverage***: It is the fraction of the total number of c-uses that have been covered. A c-use is defined as a path through a program from each point where the value of a variable is defined to its computation use, without the variable being modified along the path.

***P-use (Predicate) coverage***: It is the fraction of the total number of p-uses that have been covered. A p-use is a path from each point where the value of a variable is defined to its use in a predicate or decision, without modifications to the variable along the path.

Method, block, branch, statement are most widely employed levels for coverage based testing. The coverage analysis tools are language dependent. Coverage analyzers work by adding probe instructions in the program which increment counters.

These instructions are collectively called as instrumentation. For Java, according to instrumentation technique used coverage analyzers can be categorized as:

1. Source code instrumentation
2. Bytecode instrumentation
3. Those that run the code in a modified JVM.

Bytecode instrumentation adds instrumentation directly to the bytecode. It is very useful used in situations where the source code is not available. It does not require a modified VM, and more efficient as it does not require to recompile all the source code twice. It can provide integral coverage for single statement at source code level. It can be easily integrated to run on the fly.

The following are some of the most widely used coverage tools for Java.

- *Hansel:* Hansel provides code coverage for JUnit testing frame work[11].
- *Quilt:* Quilt works by instrumenting compiled classes before they are loaded into JVM. It measures block and branch coverage by maintaining counters in the instrumented code. It uses a class synthesizer to build a class at run time from a description, rather than loading it from disk and has tools for generating and modifying method control flow graphs[19].
- *Clover:* Clover allows you to control the instrumentation and coverage recording process, using source level directives, regular expression based filters and runtime system properties. It has detailed coverage reporting of Method, Statement, and Branch coverage with reporting in HTML, PDF, XML or a Swing GUI. Historical charting of code coverage and other metrics also provided[8].
- *Emma:* EMMA reports coverage by instrumenting classes either statically or on the fly. Coverage stats are aggregated at method, class, package, and "all classes" levels and reports can be generated in plain text, HTML, XML. Generated reports can highlight items with coverage levels below user-provided thresholds. EMMA can instrument individual .class files or entire .jars[9].

| Coverage Tool | Type | Types of Coverage | Instrumentation type | Works with | Requirements |
|---|---|---|---|---|---|
| Hansel | Open Source | Statement | ClassLoader | JUnit, Ant | BCEL 5.0 and JDK1.4 |
| Quilt | Open Source | Block and branch | ClassLoader | JUnit, Ant | JDK 1.4 |
| Clover | Closed Source (free for Open Source projects) | Statement, branch and method | Source code +bytecode | Integrated plugins for IDE's, Ant,Make | no external library dependencies |
| Emma | Open Source | Line,block, method, class | instruments either offline/onfly | Ant and Make | no external library dependencies |

Table 1.1: Comparison of Java coverage tools

## 1.2 Testing with Code Changes

Bug-fixes, code re-factoring, adding and modifying code are most frequent activities in software development because of which software keeps on changing. Typically new classes and methods get added/modified/deleted when software changes from one version to another version. Keeping track of changes made to the programs helps in program maintenance, test case maintenance and thus minimizes the testing effort. Changes occur when code is changed, bugs are fixed, re-factoring is performed or optimizations done on the code.

When a program changes, it is necessary to test every change to assure the correctness of the program. When program is modified, identification of test cases that must be re-executed is the basic issue. One approach is that, every time a change occurs, the entire test suite is rerun which is expensive and consumes much time. It is desirable to identify the test cases that *exercise* the changes and just reruns them. However, the optimum effort needed to retest a set of classes that have changed cannot be easily computed. Some clues can be derived from changes and previous testing coverage by keeping a track on the changes on code or the test case.

Tracking of code changes and test cases has important application with Regressions testing. When the code changes occur, it becomes very important to see the compliance of the new code. It has to be ensured that no new bugs are introduced

and the original functionality of the system is kept intact. This is done in regression testing, wherein partial testing of changed code is done instead of redoing the entire system using regression test suite.

Typically, the regression test suite is rerun on the modified software after every modification and the outcome compared with expected behavior. Such a regression testing is applied at unit level (typically, a class), cluster level (collection of classes).

Regression test case selection consists of three basic steps. Get the test cases for each line segment for the original code. Use version control tool or diff utility to find the changes between original and new code. And finally, select the test cases which execute line segments which are now changed[6].

This method is inefficient because it considers line segments for change analysis and does not consider control flow of the program. If a test case executed a line segment it may not necessarily execute it again after code is changed because of control flow change. Depending on the changes, it would be efficient to prioritize the test cases. Hence, a better approach would be to consider the control flow exercised by tests to obtain better tests.

Test case prioritization schemes are used to order the test cases so as to improve the effectiveness of overall testing effort. The use of prioritization can be viewed in different ways. First, it can be used to improve the error detection capability of the testing, which is one of its goals. Identifying the defects in the system quickly, gives the developer an ample scope for correcting it.

In another view, the test case prioritization scheme can potentially improve the regression testing results using the test suite, already built[10]. In our analysis framework, we capture execution details of the program with its test suite, which can be further used for ordering the test cases. This ordering is done in such a way that modifications in the program are focused in the regression testing.

There are various approaches for test case prioritization, most of which use coverage information. These techniques are based on total coverage of code components, which include method,block,branch and MCDC coverage[1]. Specific algorithms are designed which use the required coverage information like uncovered blocks, for prioritization fulfilling an objective function.

## 1.3 Organization of the Thesis

This work consists of two parts: an instrumentation and change analysis. In the first, we have developed a coverage analyzer which instruments the code so as to collect different types of coverage data. The focus is laid on class, method, branch, block and predicate coverages. For each of these, the class is instrumented on bytecode to record the necessary information. The instrumentation engine has a consistent mapping with test cases which allows us to do test case level analysis. The second part consists of change analysis wherein, the required test cases for executing the changed part of the code are identified and prioritized.

This report is organized as follows. In chapter 2, we describe the detailed bytecode analysis and instrumentation details. Chapter 3 contains the change analysis along with test case prioritization. Chapter 4 describes some implementation issues followed by some experiments and observations in Chapter 5. Conclusions and future work are given in Chapter 6.

# Chapter 2

# Overall Design and Instrumentation for Coverage

In this chapter we describe the overall design, instrumentation and database organization. For collecting coverage data, we instrument the bytecode rather than the source code. The key advantage of this approach is that it does not require the language grammar and does not require the source code.

## 2.1 Overall Design

The basic design goals are:

1. *Platform independent framework for static and dynamic analysis*: The framework should work on all platforms irrespective of compiler and optimizations used. It should support all kinds of applications such as multi threaded programs, applets, servlets etc.

2. *All the modules should work independent of each other*: Instrumentation, Coverage, Change analysis and Test case prioritization should be able to work independently or together. It should be able to integrate with version control systems with minimum effort.

3. *Provide test case level analysis also*: This is important for debugging as well as test case analysis and design. The tester should be able to define a test case. For example, test case coverage for JUnit[14] test cases is supported by coverage tool.

4. *Persistent coverage and change history*: Coverage data and test case information should be maintained persistently in a database. Even when multiple users use single database, coverage information should be consistent.

5. *User friendly environment*: User friendly environment to do selective coverage analysis at class and test case level. To give class, method, block, branch and predicate coverage for class and test case level. Tester should be able to inspect the coverage while the program is running for debugging purposes.

6. *Instrumentation and runtime overhead*: Instrumentation overhead and execution overhead should be as minimum as possible.

The overall system has four parts. The architecture is shown in Figure 2.1.

- Instrumentor.
- Change analyzer.
- Test case prioritization.
- Database.

Code under test(CUT) is given to the instrumentation engine which inserts the probes at appropriate positions depending on coverage type. The instrumented code is then run with a proper test suite. During the execution, when a probe is invoked, the necessary data is updated to the database. The database also records overall coverage and execution information for each test case.

When the program is changed, the changes are tracked during change analysis. Here, a decoder is used to identify the changed code. The change analyzer is responsible for updating the coverage information so as to reflect the changed program. Finally, the test case prioritization ranks the effected test cases so as to achieve

some objective function like getting maximum coverage with minimum number of test cases.

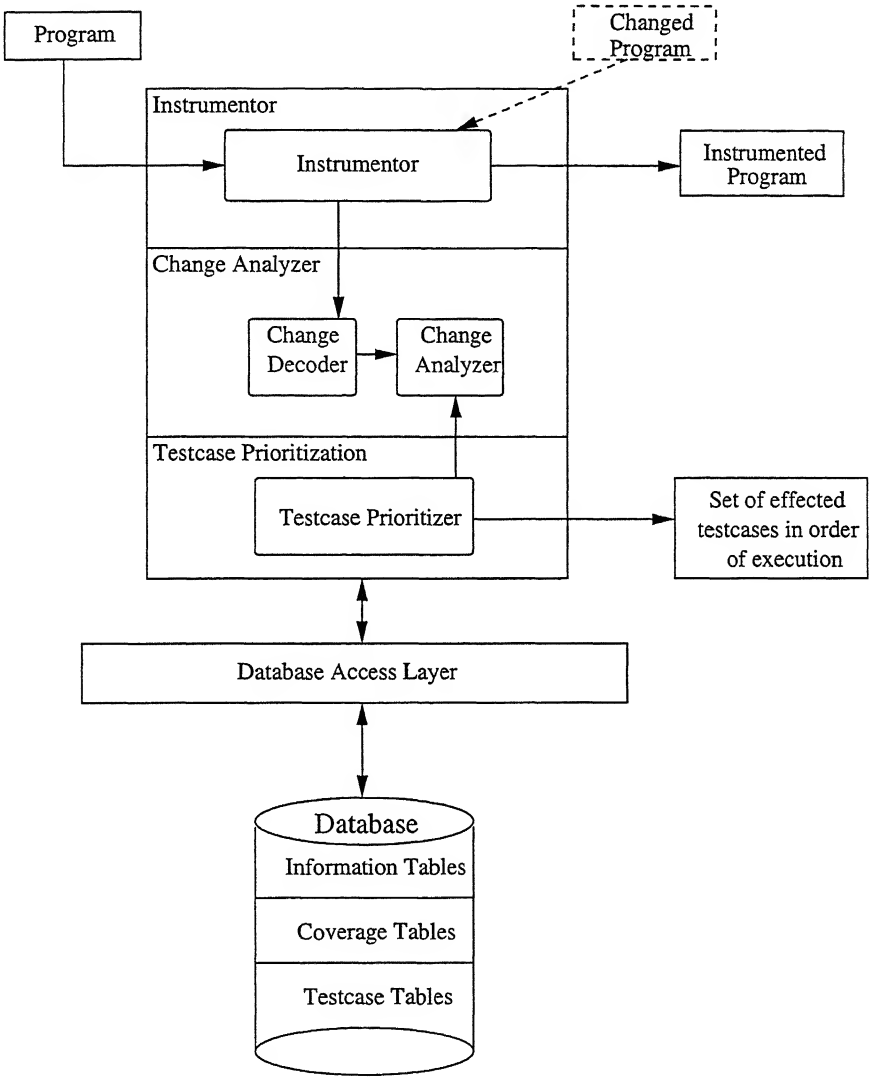



Figure 2.1: Overall Architecture

In this chapter we describe instrumentation and how the database is organized. In Chapter 3 we describe change analysis and test case prioritization.

## 2.2 Instrumentation and Coverage

Instrumentation is done at class level and at bytecode level. It is done in two passes as shown in Figure 2.2. In first pass the bytecode of a class is analyzed, control flow graph for each class is generated and appropriate probe locations for all types of coverage are identified. In second pass the program is instrumented using probes calculated in first pass.

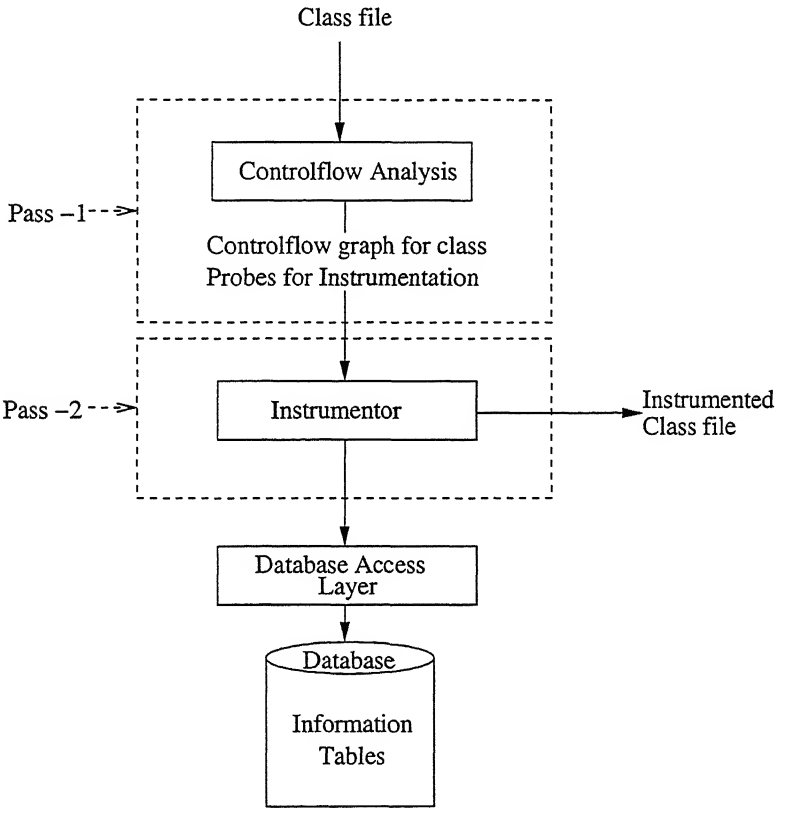



Figure 2.2: Instrumentation Passes

In this section we discuss how the control flow analysis is done at class level, how the probes for different coverages are calculated and how class is instrumented for coverage.

***Control Flow analysis***: Control flow analysis for each class is done separately. Control flow graph for class consist of set of methods in the class and calls between

them and control flow graph for each method. Control flow graph for method consists of set of basic blocks and branches between them. Inter method calls and calls to external methods are identified. Call references out side the class are referred to as external call. To decrease the complexity, method calls to java language methods are ignored. Each basic block consists of single or multiple entries and only one or two exits. Each class is represented in the form of control flow graph for static and dynamic analysis.

***Instrumentation for coverage***: Four types of coverage namely class, method, block, branch and predicate coverage were considered. During instrumentation phase, probes are inserted which call static methods of execution monitors maintaining the coverage information.

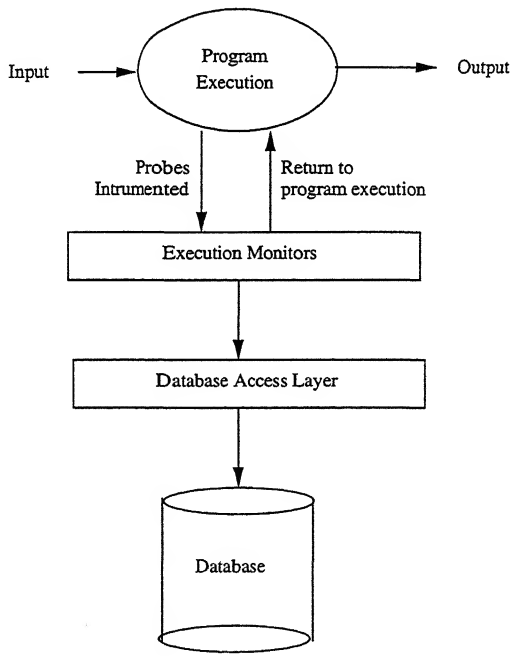



Figure 2.3: Instrumented Program Execution

Execution monitors consist of static methods which update coverage information while instrumented program is running. Whenever a call to static method of execution monitor is made, the information about the call is buffered and call is

returned immediately. Later the buffered information is transferred to database using database access layer. Database access layer is a set of classes which stores and retrieves the information from database persistently. The way in which probes are inserted into the program depends on the type of coverage. Here is the brief description of places at which probes are inserted for each coverage type.

- *Method coverage*: At start of each method a probe is inserted to track coverage information for method coverage. Probe consists of class-id and method-id.
- *Block coverage*: A probe is inserted before branch instruction of every block to track coverage information. This probe consists of class-id, method-id and block-id. All the branches to the branch instruction are redirected to the probe instruction.
- *Branch coverage*: Probes are inserted before every branch target to track branch coverage. Branch coverage of exceptions in the programs are handled separately. All the branches to the branch instruction are redirected to the probe instruction. Branch probe consists of class-id, method-id and branch-id.
- *Predicate coverage*: In first pass, all the predicates are identified. In second pass, a probe instruction is inserted to track the operands for predicate. A probe instruction is inserted before every simple predicate and branches are set appropriately. Probe consists of information about predicate-id, method-id, class-id and instructions to duplicate operands on the stack for the predicate.

## 2.3 Database Organization

Database is used to record all the coverage and test case execution information. Database organization can be categorized into three three kinds of tables.

1. Information Tables: These tables capture full information about the program such as methods, blocks, branches and predicates present in the program. It maintain instrumentation and change information about the instrumented

classes including line number information, offset in the bytecode and indices which are used by the other tables.

2. Coverage Tables: These tables record over all coverage information for all kinds of coverage and updated dynamically with execution of test cases.

3. Test case Tables: These tables consist of test case information and execution details. Test case definition consists of date of test case creation, tester, command used, input details and comments provide by the tester or developer. Each execution or run of the program is considered as a test case. Coverage for each test case and overall coverage can be obtained separately. Coverage information and execution trace for each test case is also stored in these tables.

# Chapter 3

# Change Analysis and Test Case Prioritization

In this chapter we describe how changes are analyzed and how the test cases are prioritized. Change analyzer consist of change decoder, change analyzer and impact analyzer as shown in Figure 3.1. Change decoder finds the changes made to the code. These changes are decoded by change analyzer and then their impact is analyzed by the impact analyzer.

## 3.1 Change Decoder

Change decoder finds the changes made to the code. This is done at two levels: source code level and bytecode level.

***Source code level difference:*** Source code level difference finds out the set of line numbers which are added/deleted/modified between two versions of java programs. Source code level difference is used to decrease the complexity of bytecode level difference between two classes.

***Bytecode level difference:*** Bytecode level difference is used to find out the differences between two versions of bytecode of a class. Comparison of two versions of class files at bytecode level takes O(m*n*log(n)) time where m is the number of methods and n is the number of bytecode instructions in the method. To decrease

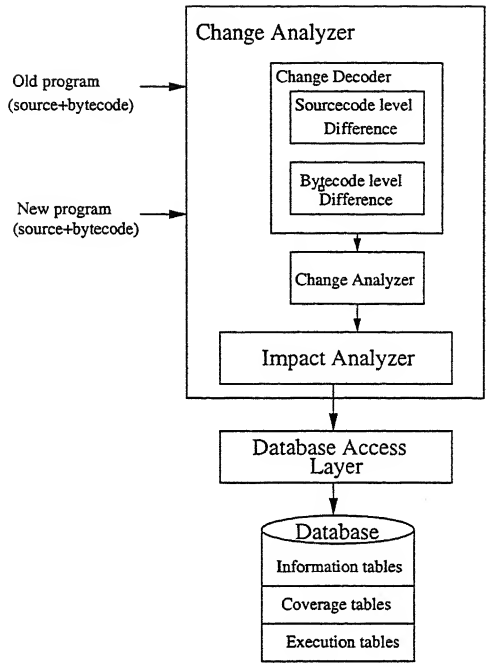



Figure 3.1: Change and Impact analysis

the complexity and increase the accuracy source code level difference is used as aid for byte code level differencing. Line numbers added/deleted/modified computed using source code level differencing used while comparing classes at bytecode level.

In bytecode level changes are analyzed at class, method, block, branch and predicate level. Bytecode of two versions of classes are taken and their bytecode are compared to find out the changes. These changes are mapped with the previous version. Every change is categorized as added/modified/deleted corresponding to previous version.

## 3.2 Change Analysis

In change analysis decoded changes are analyzed and categorized as shown in Table 3.1. Whenever a class is changed the class and the package in which it contains

is considered as changed. Whenever method is changed the method and the corresponding class and package for the class is considered as changed. Whenever a block/branch/predicate is changed the corresponding method and class is considered as changed.

| Change | Package | Class | Method | Block | Branch | Predicate |
|---|---|---|---|---|---|---|
| Added | A new package is added | A new class is added | New method is added to class | A new block is added to the control flow of method | New branch condition is added in the control flow | New predicate is added |
| Deleted | A package is deleted | A Class is deleted | Existing method is deleted | An existing block in the method is is deleted | Existing branch condition is deleted | Existing predicate is deleted |
| Modified | A class in package is modified or some classes added or deleted or modified | A class is modified or base class for the class or inheritance changed | Part of the method is changed | Instructions in the block modified /added /deleted | Conditions relating to the branch or branch target changed | Predicate or its operands changed |

Table 3.1: Change categorization

## 3.3 Impact Analysis

Impact analyzer analyzes the impact of changes made. It resets the coverage appropriately depending on changes. It also marks the execution details for test cases invalid which get affected by the changes.

### *Impact on Coverage:*

After change the present coverage for the new version depends on changes made. The coverage of the changed instructions is reset i.e initialized to zero. The coverage information for unchanged part of the code is retained. Deeper level of coverage

resetting is also possible in which control flow for the method is considered and coverage information for the units which are successors of changed units in the control flow are reset to zero. This approach of coverage resetting will have a big impact on coverage because if first block of the method is changed coverage information for all the blocks should be reset to zero. For simplicity we only reset the coverage for the changed units.

Method coverage for the method is reset to zero when part of the code of a method is changed. When a method is changed the impact analyzer analyzes changes made to method at block, branch and predicates level. Depending on the change, it reorganizes the block structure for changed method. Subsequently, the reorganized blocks are reflected in execution and coverage tables. Coverage is retained for the blocks which are unmodified and the coverage for the blocks which are got changed is reset to zero. Depending on the block changes coverage for branches which lead to the block changes. Effected branches are identified and marked as modified. Coverage for all modified branches is reset to zero. Coverage for all the predicates which were affected by the change is reset to zero and predicates are marked as changed.

### *Impact on test cases:*

Test cases which executed the modified code in the older version of the code gets effected by the change. After change test case execution information is modified to current version. This is done by mapping the indices in test case tables of previous version to the new indices for new version of the code. All the invalid indices are marked negative. There are two approaches for resetting coverage based on test case execution information.

1. Decrement the coverage count for the units which are executed by the test case after encountering very first change because the test case may follow different path after that change and execution information after that change is invalid.

2. Retain the coverage information for the units which are executed after changed part of the code. Only reset the coverage for the changed part.

## 3.4 Test case Analysis and Prioritization

After a change, test cases which executed the modified part of the code are affected. Some test cases become invalid and some test cases need to be re-executed. Test case prioritization is a process of selecting test cases from set of affected test cases so as to achieve testing criteria objective. In test case prioritization the goodness of the test case is calculated from previous coverage and change information and test cases are arranged in order of goodness of the test case.

### *Algorithm for test case prioritization:*

Test case prioritization consists of three steps:

1. Identify test cases that were effected by the changes.
2. Identify invalid test cases from effected test cases.
3. Prioritize affected test cases which are valid based on objective function.

***Identify test cases that were effected by the changes:*** If a test case previously executed a modified or deleted block of the code it is considered as affected test case. For all the added blocks if a test case executes its predecessor and any of its successor it is considered as effected test case. If the block is of return/exit type then any test case which executed its predecessor in the control flow is considered as effected. If the block is at the start of the method then all the test cases which previously executed the method are considered as effected.

***Identify invalid test cases:*** Any test case which has executed deleted method as its start of execution is considered as invalid test case.

***Prioritize effected test cases:*** Affected test cases are taken and probability is assigned to each test case depending on the type of changes it is likely to execute. Each test case is given probability based on how many changes it is likely to execute.

Let $P$ be the probability function expressed for various parameters like test case, method, changed method etc. We have designed the following heuristic for prioritizing the test cases.

$$P(testcase) = \frac{\sum P(changed\ methods\ the\ test\ case\ executes)}{No.of\ changed\ methods} \quad (1)$$

$$P(method) = \frac{\sum P(changed\ blocks\ in\ the\ method)}{No.of\ changed\ blocks\ in\ the\ method} \quad (2)$$

$$P(block) = \frac{Degree(block)}{maximum\ Degree(block)\ in\ the\ method} \quad (3)$$

where degree of block is given as:

$$Degree(block) = Total\ no.of\ branches\ to\ block + branches\ to\ changed\ blocks\ from\ the\ block \quad (4)$$

For each test case the above probabilities are calculated and they are ranked in the order of their probabilities for prioritization.

# Chapter 4

# Implementation

Implementation is done in Java using BCEL(Byte Code Engineering Library)[1]. MySql [2] was used for maintaining database. This chapter briefly describes the implementation of the tool.

## 4.1 Instrumentation for Coverage

Instrumentation is done at bytecode level for each class. Instrumentation can be done for method, block, branch and predicate coverage either individually or selectively. This can be done for multiple classes or package as a whole. This process is carried in two passes. In the first pass probe location for instrumentation are identified. In second pass probe instructions are inserted at identified location to track coverage details. In this section we describe how probes are located for each coverage type and what are the instructions inserted to track the coverage information.

Every class is assigned a unique number called class-id before instrumenting. The coverage, execution and information tables in the database are referenced with this class-id. Information about the class like version, date of instrumentation, package, change information is stored in the database.

[1]BCEL:http://jakarta.apache.org/bcel/
[2]Mysql:http://www.mysql.com/

## Instrumentation for method Coverage:

Every method in the class is given a unique number(method-id) to identify it within the class. At start of the bytecode of the method probe instructions are added to collect the coverage information so as to ensure that, before executing the method coverage information is stored. Probe instruction consists of invocation of static method which takes a string as argument containing concatenated class-id and method-id.

When instrumented probe is invoked, the database entries corresponding to the method are updated suitably.

## Instrumentation for block coverage:

Blocks are decided based on branch instructions. Any control flow instruction at bytecode level is considered as end of block. All targets of control flow statements are considered as start of new block.

***A block consist of***:

- Block id which is unique for a method in a class.
- Starting and ending offset in the bytecode.
- Starting and ending line in the source code.
- Target block in case of control flow.
- Type of control flow (internal/external/return/exit).
- Change information (Added/deleted/modified/unchanged).

In first pass, bytecode of the method is parsed and the method is arranged as set of blocks and branches between them.

***Instrumentation procedure for coverage:***

To track block coverage, probe instructions are added before branch instruction or at the end, if the block does not contain any branch instruction. If branch instruction is the only instruction in the block then all the branches leading to the

branch are redirected to instrumented probe. All exception targets are appropriately reset if altered by instrumented probes.

During execution when instrumented probe is invoked, coverage information pertaining to class-id, method-id and block-id is stored in the database.

## Instrumentation for branch coverage:

Every branch in the method is identified by class-id,method-id and branch-id(offset of branch instruction in the bytecode). Any control flow statement at bytecode level is considered as branch instruction. At bytecode level, branches can be categorized as

- Conditional
- Unconditional
- Exceptions

Every conditional branch has two branches. One is branch target and other is its succeeding instruction. Two probes need to be added to track branch coverage information. For unconditional branches only one probe is used to track coverage information as the branch is always takes place. Exceptions and throws are separately handled. Java classfile maintains exception table. For each exception, possible targets are taken from exception table. A probe is inserted before exception target to track the coverage information for that particular exception. Targets in the exception table for the exception are reset to instrumented probe.

## Instrumentation for predicate coverage:

Predicate coverage checks whether every possible value for a predicate is taken and the predicate executed to produce every possible output.

In first pass all the simple predicates and compound predicates in the method are found. Probe instructions are added before every simple predicate and just after predicate operands are loaded on to the stack. Instrumented probe instructions

contain instructions to duplicate the operands of the predicate on the stack and static method call to execution monitors. Execution monitors pop the operands from the stack and evaluate the predicate and store the operands, evaluated result in database. This is shown in Figure 4.1.

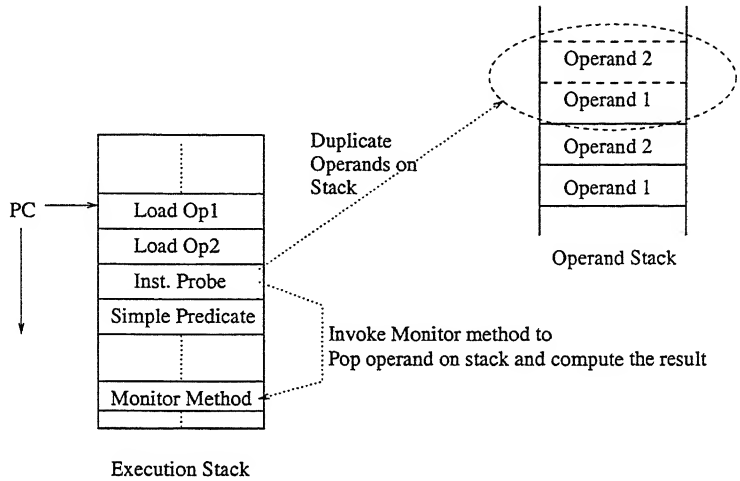



Figure 4.1: Instrumented predicate on execution

Every simple predicate will have one or two operands. These operands can be of BOOLEAN, INTEGER, FLOAT, LONG or DOUBLE type. For object comparisons only out come is considered. Predicate returns two kinds of values: BOOLEAN or INTEGER. All equality operators return TRUE or FALSE except comparison of LONG values which return 0, 1 or -1 depending on the output of the condition. Coverage measure involves checking some predicate for TRUE or FALSE and some predicate for 0, 1 or -1 depending on the opcode of the predicate.

## 4.2 Database Implementation

Database is maintained using MySql. Jdbc is used to store and retrieve information from MqSql database. Database can be maintained centrally and different developers/testers can use single database over network.

Operations that can be performed on database are:

**On information tables:**

1. Store information about instrumented class.
2. Get information about instrumented class.
3. Update change and version information.
4. Delete information about a class.

**On coverage tables:**

1. Update coverage information.
2. Get coverage information for a class or overall.
3. Reset coverage for particular class.
4. Delete coverage information for particular class.

**On test case tables:**

1. Store execution information for particular test case.
2. Get execution information for particular test case.
3. Get coverage information for particular test case or set of test cases.
4. Delete test case execution information.

## 4.3 Change Analysis and Test Case Prioritization

### Change Analysis:

*GNU diff* utility is used to find out the line differences between two versions of java programs at source code level. If version control system is available then the version control system can be used to find of the changes occurred from one version to another. Bytecode of two versions of the classes are parsed and compared to find

the difference at class, method, block, branch and predicate level. All the methods in new version are compared with the methods in old version. If part of code of method in the new version is not matched with old version then it is considered as modified. If a method in new version is not found in old version, it is considered as added method. For all methods which have added and modified a block, branch and predicate level difference is done. Changed information is used by the impact analyzer for resetting coverage and execution information.

### Test Case Prioritization:

Test case prioritization is implemented using the test case information present in the database. Affected test cases are found by querying the database. Execution tables are inspected for these test cases. All effected methods, blocks, branches, predicates and the test cases which executes the changed units are found by simple sql query. Change information is retrieved using the change analyzer. Test cases are given the probability as defined in the algorithm and are ranked in order of their probability.

## 4.4 GUI

The tool set provides a user friendly GUI implemented using JAVA Swing. It is implemented as separate module as shown in Figure 4.2. GUI uses Database Access Layer to retrieve coverage and test case information from the database. The required information on all the coverage measures is displayed at package level and class level. The coverage information can be seen for the entire test suite as well as for individual test case. A snapshot of the tool's GUI is shown in Figure 4.3.

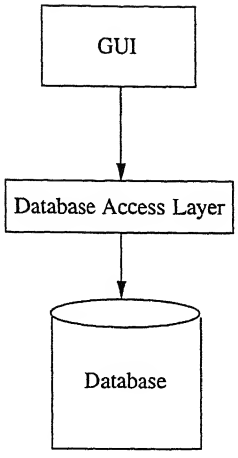



Figure 4.2: Graphical User Interface Design

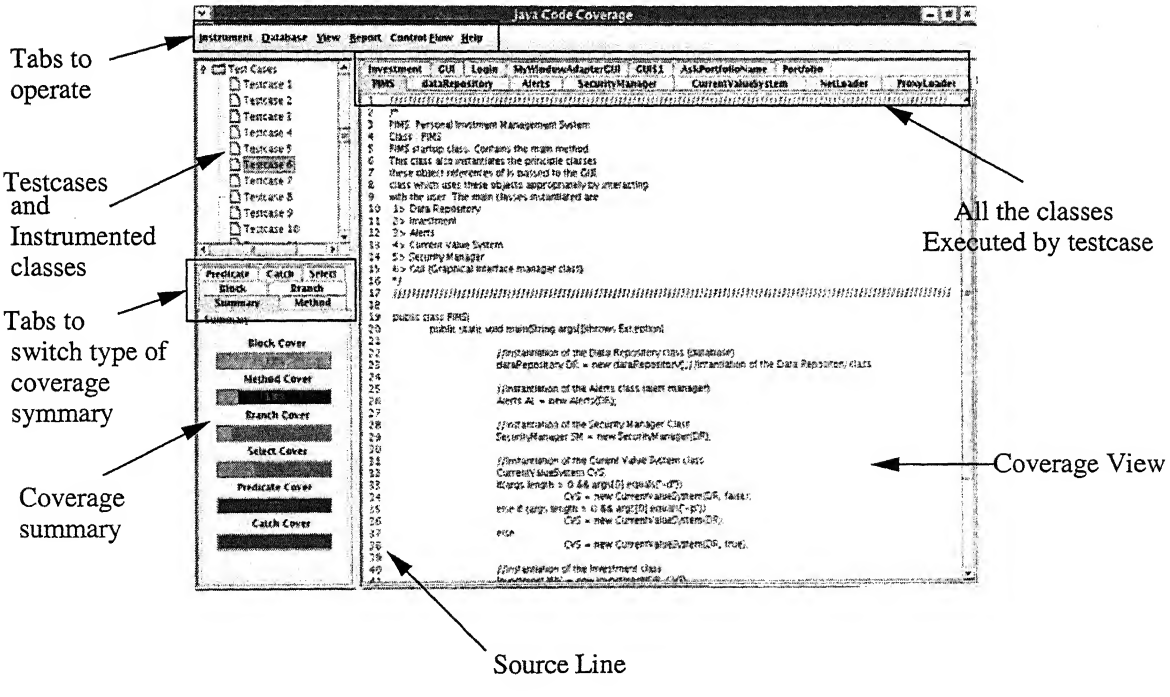



Figure 4.3: Snapshot of Graphical User Interface

# Chapter 5

# Experiments

## 5.1 Overhead

### *Instrumentation Overhead:*

Instrumentation overhead depends on the number of blocks,branches and number of predicates present in the program. Table 5.1 shows the instrumentation overhead for sample programs.

| Software Package | NCLOC | Size before instrument | Size after Instrument | Time taken to instrument | Number of classes | Number of methods | Number of blocks |
|---|---|---|---|---|---|---|---|
| JRisk | 6500 | 868.5KB | 1MB | 82423ms | 114 | 5944 | 1229 |
| MegaMek | 40000 | 516.1KB | 712KB | 181610ms | 270 | 3314 | 16468 |
| PIMS | 3000 | 129.2KB | 219KB | 4816ms | 40 | 242 | 1188 |

Table 5.1: Instrumentation overhead for few softwares

- Size overhead : Approximately increases by half of original when instrumented for all coverages.
- Time for instrumentation : Approximately takes 4ms for each block to instrumenting for all coverages.

## 5.2 Experiments

### *Bug Identification with Coverage based testing*

In this experiment we have taken simple hotel management program. It consist of six classes. Twenty bugs are inserted in the program. Table 5.2 shows the type of bugs inserted and how many are identified by coverage type. In Figure 5.1 we plot a graph between the number of bugs identified by each coverage type and bugtype.

*Type of inserted bugs in the program*

| Bug | Details | Total No. of inserted bugs |
|---|---|---|
| URC | Unreachable code | 1 |
| IIO | Initialization operator | 2 |
| LCO | Literal Change Operator | 3 |
| LOR | Language Operator Replacement | 3 |
| ACO | Argument Change Order | 1 |
| MNR | Method Name Replacement | 1 |
| VRO | Variable Replacement Operator | 3 |
| SSO | Statement Swap Operator | 1 |
| CFD | Control Flow Disruption | 3 |

***Bugs detected by coverage type***

| BugType<br>Coverage Type | URC | IIO | LCO | LOR | ACO | MNR | VRO | SSO | CFD | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| Total-Bugs | 1 | 2 | 3 | 3 | 1 | 1 | 3 | 1 | 3 | 18 |
| Block Coverage | 0 | 0 | 2 | 2 | 0 | 1 | 3 | 1 | 2 | 11 |
| Branch Coverage | 1 | 1 | 3 | 2 | 0 | 1 | 3 | 1 | 2 | 14 |
| Predicate Coverage | 1 | 1 | 3 | 3 | 0 | 1 | 3 | 1 | 3 | 16 |

Table 5.2: Type of bugs inserted in the program

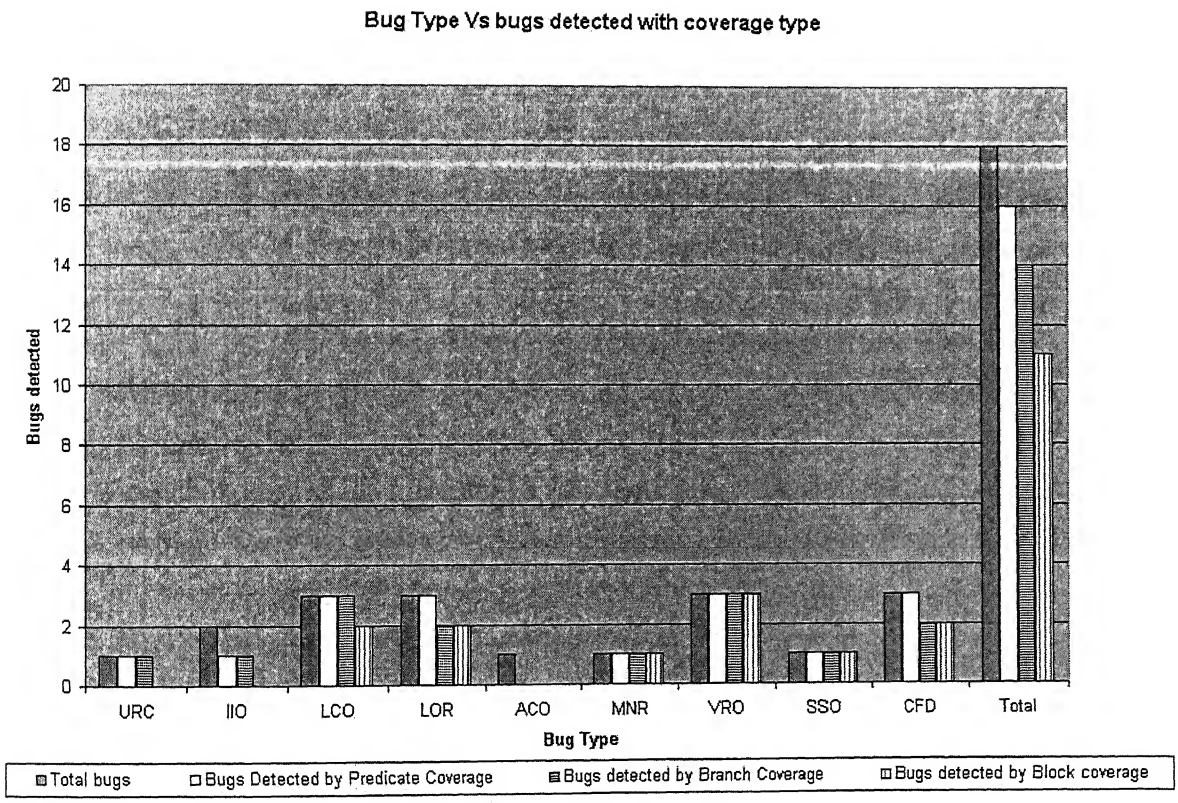



Figure 5.1: Bugs detected by coverage type

| Software | PIMS |
|---|---|
| LOC | 4000 |
| Number of Classes | 40 |
| Number of Methods | 242 |
| Number of Blocks | 1188 |
| Number of Branches | 1340 |
| Number of Catches | 51 |
| Number of simple predicates | 333 |
| Time taken for instrumentation | 4816 ms |

Table 5.3: PIMS details

### *Coverage Increases with Testing*

We conducted some experiments to study how the coverage increases with number of test cases. In the first experiment we have taken PIMS(Personal Investment Management System) and studied behavior of coverage as the number of test cases increases. Table 5.2 shows the software details and time taken to instrument PIMS.

We have executed 57 distinct test cases from the test plan. Figure 5.2 shows how different coverages increase with number of test cases.

We inspected growth in branch coverage by taking ten different series of test cases. Coverage for ten different series of test cases is shown in Figure 5.3. The average of these ten series of test cases is considered which is shown in Figure 5.4.

Also from coverage information we identified deadcode and suggested test cases to reach 85 percentage coverage. Table 5.4 shows the block coverage percentage for each class. Emphasis is laid on classes which has less than 80 percentage coverage.

In the second experiment we took Jakartha RegExp java package and studied behavior of coverage as the number of test cases increases. Table 5.5 shows the package details and the time took to instrument the package.

We have executed 215 test cases. We inspected how coverage increases by taking ten different series of test cases. Figure 5.5 shows the coverage for each test case series. To generalize we took the average of ten different coverage values and plot the curve as the number of test cases increases. Figure 5.6 shows the average coverage as the number of test cases increases. A logarithmic curve is fitted based on the results from coverage analysis and is shown in the figure.

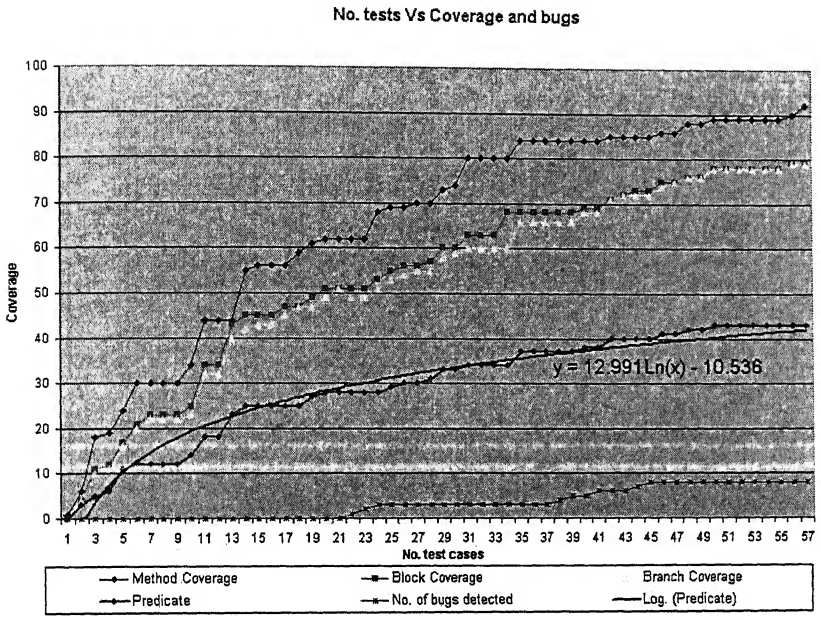



Figure 5.2: Coverage growth with number of test cases for PIMS

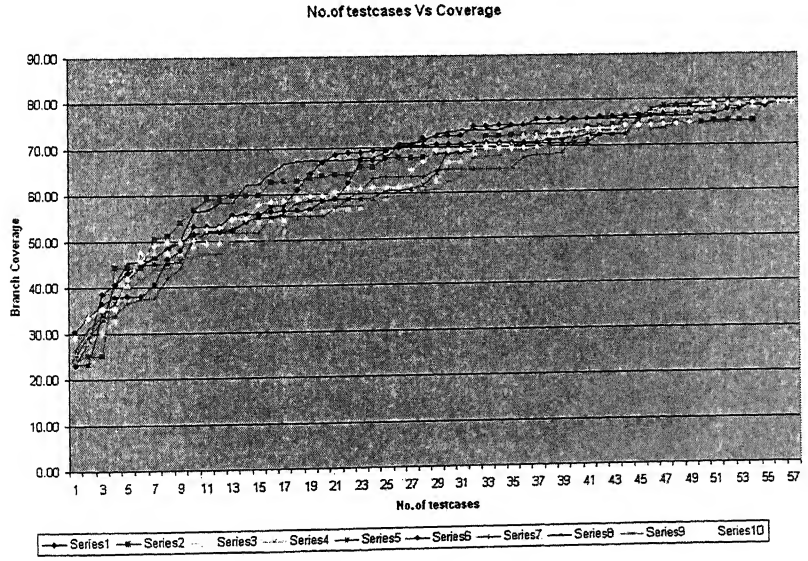



Figure 5.3: Coverage for ten different series of test cases for PIMS

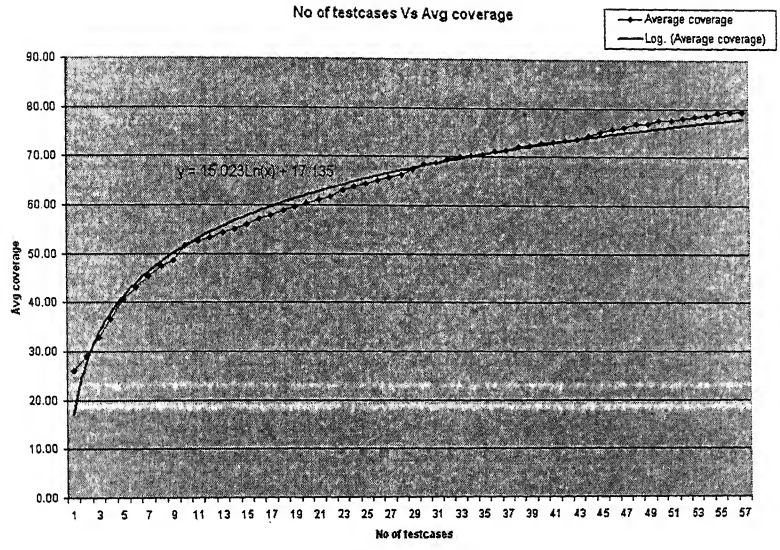



Figure 5.4: Average coverage of ten different test case series for PIMS

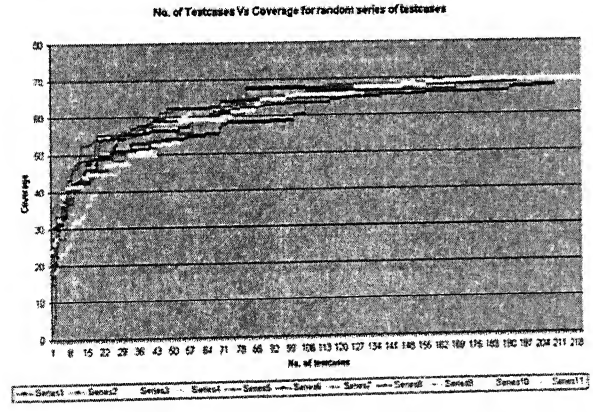



Figure 5.5: No. of test cases Vs coverage for RegExp

| Class | Total Blocks | Not Covered Blocks | Coverage Percentage |
|---|---|---|---|
| About | 19 | 3 | 84.21 |
| Alerts | 37 | 0 | 100.00 |
| AskBankEditTransactionDetails | 24 | 2 | 91.67 |
| AskBankTransactionDetails | 21 | 2 | 90.48 |
| AskNewPrice | 11 | 4 | 63.64 |
| AskPortfolioName | 28 | 2 | 92.86 |
| AskSecurityName | 60 | 4 | 93.33 |
| AskShareEditTransactionDetails | 28 | 4 | 85.71 |
| AskShareTransactionDetails | 24 | 3 | 87.50 |
| ChangePassword | 16 | 5 | 68.75 |
| CurrentValueSystem | 27 | 11 | 59.26 |
| dataRepository | 135 | 8 | 94.07 |
| DeleteAlert | 15 | 2 | 86.67 |
| EditPrices | 14 | 3 | 78.57 |
| Error | 4 | 0 | 100.00 |
| FileDownload | 30 | 10 | 66.67 |
| GUI | 161 | 21 | 86.96 |
| Help | 3 | 0 | 100.00 |
| Installation | 38 | 12 | 68.42 |
| InstallFinal | 3 | 1 | 66.67 |
| InstallNext | 58 | 28 | 51.72 |
| Investment | 58 | 10 | 82.76 |
| Login | 15 | 2 | 86.67 |
| Message | 4 | 0 | 100.00 |
| MyAuthenticator | 4 | 2 | 50.00 |
| MyWindowAdapterGUI | 2 | 1 | 50.00 |
| NetLoader | 39 | 38 | 2.56 |
| PIMS | 9 | 3 | 66.67 |
| Portfolio | 102 | 25 | 75.49 |
| ProxyLoader | 22 | 3 | 86.36 |
| security | 117 | 14 | 88.03 |
| SecurityDS | 3 | 0 | 100.00 |
| SecurityManager | 11 | 1 | 90.91 |
| SetAlerts | 19 | 3 | 84.21 |
| ShowAlerts | 9 | 0 | 100.00 |
| Transaction | 10 | 0 | 100.00 |

Table 5.4: Block coverage report for PIMS

| Package | RegExp |
|---|---|
| Number of Classes | 14 |
| Number of Methods | 110 |
| Number of Blocks | 754 |
| Number of Branches | 862 |
| Number of Catches | 346 |
| Number of simple predicates | 329 |
| Time taken for instrumentation | 6160 ms |

Table 5.5: RegExp details and instrumentation time

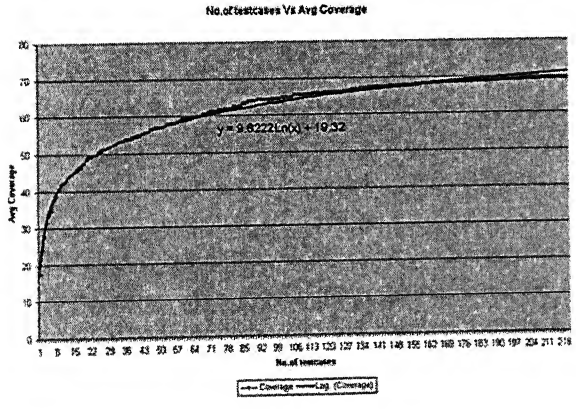



Figure 5.6: No. of test cases Vs average coverage for RegExp

### *Effect of Versions*

In this experiment four versions of Jakartha's RegExp package are taken and changes are analyzed after each version change. It consist of 143 test cases for versions 0,1 and 2 and 215 test cases for version 3. 5.6 shows the RegExp package details and time taken to instrument.

| Package | Jakarta RegExp |
|---|---|
| Number of Classes | 10 |
| Number of Methods | 179 |
| Number of Blocks | 727 |
| Number of Branches | 699 |
| Number of Catches | 8 |
| Number of selects | 213 |
| Number of simple predicates | 277 |
| Time taken for instrumentation | 5582 ms |

Table 5.6: Instrumentation information for RegExp V.1.0

Table 5.2 shows how coverage varies after each version change. After each version change impact on coverage is inspected and depending on changes test cases are prioritized to get maximum coverage with minimum number of test cases. Results show coverage stabilizes after executing 30 percent of prioritized test cases.

**Coverage after executing 148 test cases on RegExp V.1.0**

| Coverage type | No.of units | No.of Covered units | Coverage Percentage |
|---|---|---|---|
| Class | 11 | 11 | 100 |
| Method | 79 | 53 | 53 |
| Block | 727 | 451 | 67 |
| Branch | 699 | 420 | 60 |
| Select | 213 | 87 | 40 |
| Exceptions | 8 | 0 | 0 |
| Predicate | 277 | 165 | 40 |

**Coverage after version change RegExp V.1.0 to RegExp V.1.1**

| Type | No.of units | Modified units | Added units | Coverage after version change | Coverage after prioritized test case execution |
|---|---|---|---|---|---|
| Class | 14 | 11 | 3 | 79 | 100 |
| Method | 108 | 3 | 29 | 49 (53/108) | 55 (60/108) |
| Block | 801 | 130 | 74 | 40 (321/801) | 58 (467/801) |
| Branch | 764 | 143 | 65 | 36 (277/764) | 55 (426/764) |
| Select | 346 | 0 | 133 | 25 (87/346) | 39 (135/346) |
| Exceptions | 16 | 0 | 8 | 0 (0/16) | 0 (0/16) |
| Predicate | 298 | 68 | 44 | 25 (74/298) | 39 (116/298) |

**Coverage after version change RegExp V.1.1 to RegExp V.1.2**

| Type | No.of units | Modified units | Added units | Coverage after version change | Coverage after prioritized test case execution |
|---|---|---|---|---|---|
| Class | 14 | 13 | 0 | 100 | 100 |
| Method | 108 | 5 | 3 | 49 (52/108) | 55 (60/108) |
| Block | 806 | 71 | 5 | 49 (396/806) | 52 (423/806) |
| Branch | 764 | 77 | 1 | 45 (349/765) | 56 (433/765) |
| Select | 346 | 0 | 0 | 39 (135/346) | 39 (135/346) |
| Exceptions | 16 | 0 | 0 | 0 (0/16) | 0 (0/16) |
| Predicate | 298 | 3 | 0 | 25 (74/298) | 39 (116/298) |

**Coverage after version change RegExp V.1.2 to RegExp V.1.3**

| Type | No.of units | Modified units | Added units | Coverage after version change | Coverage after prioritized test case execution | Coverage after additional test cases execution |
|---|---|---|---|---|---|---|
| Class | 14 | 13 | 0 | 100 | 100 | 100 |
| Method | 110 | 23 | 2 | 34(37/110) | 55 (61/110) | 58 (64/110) |
| Block | 754 | 210 | 0 | 21 (161/754) | 52 (393/754) | 64 (487/754) |
| Branch | 862 | 151 | 0 | 21 (184/862) | 37 (323/862) | 42 (364/862) |
| Select | 329 | 0 | 0 | 41 (135/329) | 41 (135/329) | 41 (136/346) |
| Exceptions | 16 | 0 | 0 | 0 (0/16) | 0 (0/16) | 0 (0/16) |
| Predicate | 346 | 68 | 44 | 12 (41/346) | 37 (128/346) | 40 (139/346) |

Table 5.7: Coverage as version changes

# Chapter 6

# Conclusions and Future Work

Most of the coverage tools do not work at test case level and does not maintain change information. In normal testing, most of the testing effort goes in vain when changes occur to the code. Hence the process of testing has to be repeated on changed code.

In this thesis work, we have implemented class, method, block, branch and predicate coverage which is much stronger than other coverage measures. The coverage analyzer is implemented at bytecode level which is efficient than other source code instrumentation tools. Apart from this, we have designed a change analyzer which reuses the coverage and test case execution information when changes occur to the code.

Our experiments show that the coverage increases with number of test cases. With a few softwares and test suites, our experiments indicate that the coverage value follows logarithmic value of number of test cases. The coverage data and test case execution information can be used efficiently for testing the changed code.

This work has opened many directions for further extensions. Many other coverage measures can be calculated from existing coverage information without instrumenting code or parsing the class.

## Extending Coverage Measures:

From existing coverage information Condition/Decision, Relational operator, MCDC and Path coverages can be measured without re-instrumenting or parsing the class file for the new coverage measures. Here is a brief description of how to calculate the new coverage measures with the existing framework.

- Condition/Decision coverage: In Condition/Decision coverage only simple predicates are considered and their output is inspected for *true* and *false*. It only needs inspecting information from Coverage tables which does not need any extra effort to instrument or parse the class. The existing predicate coverage information can be used to do this.

- Relational operator coverage: In relational operator coverage only simple predicates which involve relational operator are considered and their output is inspected for true or false. Information tables for predicates contains opcode for every simple predicate. To measure relation operator coverage, only simple predicates which have opcode equal to relational operator are to be considered. It only needs inspecting instrumentation information and coverage data for predicates which have opcode equal to relational operator.

- MCDC coverage: The Modified Condition Decision Coverage (MCDC) reports coverage of occurrences in which triggering individual subcondition results in change of result from true to false or false to true. To measure MCDC coverage two conditions need to be checked

  1. Whether every decision and every condition within the decision resulted in every outcome.
  2. Whether every condition independently affected the decision.

  The above information can be inferred from execution and information tables for predicates.

- Path coverage: Path coverage measures the coverage of all feasible paths that the program can possibly take. All feasible paths can be calculated from

control flow graph of the program and their coverage can be inferred from execution tables of test cases for branch coverage.

The following are some coverage measures which need parsing of the class file.

- P-use coverage: As the predicates execution information is already available, P-use coverage can be computed by finding out data definitions and their corresponding predicates.

## Mutation Testing:

In mutation testing, the program is modified to create many mutants (faulty version of the program). If the test suite is adequate it must kill all the mutants i.e identify all the mutants that were created. The ratio of killed mutants to the total mutants indicates the sensitiveness of the code to the changes. The following are some kinds of mutations that are possible using our framework.

- Method based mutation: One example of method level mutation is inverting the output of method. This be done in second pass of instrumentation.

- Branches and Block based mutation: Various branch level mutations are possible using block analysis that is done at bytecode level. Our framework provides functions to manipulate branches at bytecode level; like mutate block execution, redirect branches, reverse branch condition, make conditional unconditional branch and mutate branch. Using these mutations, required mutants can be created.

- Predicate based mutation : Various kinds of mutations are possible at predicate level. For example, inverting opcode of predicate, reversing condition output, reversing operands etc. These mutations can be done at bytecode level in second pass of instrumentation.

## Predicate Based Testing:

In predicate based testing, test cases are designed to cover all possible reachable states of the program based on predicates. Tool helps in analyzing test cases for

predicate coverage. Our tool is also useful in analyzing predicate-complete testing (PCT)[4].

# Appendix A

# Byte code analysis on simple program

In this appendix explains how bytecode of java program is analyzed. Simple java program which finds the nearest prime number greater than given number is considered and its bytecode analyzed.

Block analysis and instrumentation for simple program:

```
1  package Prime;
2
3  public class Prime {
4
5  // check whether a number is prime
6  static boolean isPrime(int n) {
7
8         // 2 is the smallest prime
9         if (n <= 2) {
10                  return n == 2;
11          }
12
13 // even numbers other than 2 are not prime
14          if (n % 2 == 0) {
15                  return false;
16          }
17
18 // check odd divisors from 3
19    // to the square root of n
20    for (int i = 3, end = (int)Math.sqrt(n);i <= end; i += 2) {
21
```

```
22                  if (n % i == 0) {
23                        return false;
24                  }
25    }
26
27 return true;
28 }
29 // find the smallest prime >= n
30        static int getPrime(int n) {
31        while (!isPrime(n)) {
32                  n++;
33           }
34 return n;
35 }
36
37 public static void main(String args[]) {
38 System.out.print("Smallest prime greater than given Numer: ");
39 System.out.println(getPrime(Integer.parseInt(args[0])));
30
41 }
42
43 }
```

In first pass of the instrumentation Class files is analyzed and program is arranged in set of blocks and probes for branches, predicates are determined. Table A.1 shows blocks in the class, branches between the blocks. Information shown in the table is stored in database.

Bytecode after instrumentation for block coverage(instrumented instructions marked *):

```
static boolean isPrime(int);
  Code:
   0: iload_0
   1: iconst_2
   *    2: ldc #66; //String 1 1 1
   *    4: invokestatic #57;
   7: if_icmpgt
   10: iload_0
   11: iconst_2
```

```
*    12: ldc #68; //String 1 1 2
*    14: invokestatic #57;
17: if_icmpne
20: iconst_1
*    21: ldc #70; //String 1 1 3
*    23: invokestatic #57;
26: goto
29: iconst_0
*    30: ldc #72; //String 1 1 4
*    32: invokestatic #57;
35: ireturn
36: iload_0
37: iconst_2
38: irem
*    39: ldc #74; //String 1 1 5
*    41: invokestatic #57;
44: ifne
47: iconst_0
*    48: ldc #76; //String 1 1 6
*    50: invokestatic #57;
53: ireturn
54: iconst_3
55: istore_1
56: iload_0
57: i2d
58: invokestatic #2;
61: d2i
*    62: ldc #78; //String 1 1 7
*    64: invokestatic #57;
67: istore_2
68: iload_1
69: iload_2
*    70: ldc #80; //String 1 1 8
*    72: invokestatic #57;
75: if_icmpgt
78: iload_0
79: iload_1
80: irem
*    81: ldc #82; //String 1 1 9
*    83: invokestatic #57;
86: ifne
```

```
    89: iconst_0
    *     90: ldc #84; //String 1 1 10
    *     92: invokestatic #57;
    95: ireturn
    96: iinc 1, 2
    *     99: ldc #86; //String 1 1 11
    *     101: invokestatic #57;
    104: goto
    107: iconst_1
    *     108: ldc #88; //String 1 1 12
    *     110: invokestatic #57;
    113: ireturn

static int getPrime(int);
  Code:
    0: iload_0
    1: invokestatic #3;
    *     4: ldc #90; //String 1 2 1
    *     6: invokestatic #57;
    9: ifne
    12: iinc 0, 1
    *     15: ldc #92; //String 1 2 2
    *     17: invokestatic #57;
    20: goto
    23: iload_0
    *     24: ldc #94; //String 1 2 3
    *     26: invokestatic #57;
    29: ireturn

public static void main(java.lang.String[]);
  Code:
    0: getstatic #4;
    3: ldc #5; //String Smallest prime greater than given Numer:
    5: invokevirtual #6;
    8: getstatic #4;
    11: aload_0
    12: iconst_0
    13: aaload
    14: invokestatic #7;
    17: invokestatic #8;
    20: invokevirtual #9;
```

```
    *     23: ldc #97; //String 1 3 1
    *     25: invokestatic #57;
    28: return

}
```

| Package | prime | | Package-id | 1 | |
|---|---|---|---|---|---|
| Method Name | | | public void <init>() | | |
| Method-id | | | 0 | | |
| Block-id | Block-start | Block-end | Start-Line | End-Line | Branch Target |
| 1 | 0 | 2 | 3 | 3 | Return |
| Method Name | | | static boolean isPrime(int arg0) | | |
| Method-id | | | 1 | | |
| Block-id | Block-start | Block-end | Start-Line | End-Line | Branch Target |
| 1 | 0 | 2 | 9 | 9 | 5 |
| 2 | 3 | 5 | 10 | 10 | 4 |
| 3 | 6 | 7 | 10 | 10 | Return |
| 4 | 8 | 10 | 10 | 10 | Return |
| 5 | 11 | 13 | 14 | 14 | 7 |
| 6 | 14 | 15 | 15 | 15 | Return |
| 7 | 16 | 23 | 20 | 20 | Null |
| 8 | 24 | 25 | 20 | 20 | 12 |
| 9 | 26 | 29 | 22 | 22 | 11 |
| 10 | 30 | 31 | 23 | 23 | Return |
| 11 | 32 | 33 | 20 | 20 | 8 |
| 12 | 34 | 35 | 27 | 27 | Return |
| Method Name | | | static int getPrime(int arg0) | | |
| Method-id | | | 2 | | |
| Block-id | Block-start | Block-end | Start-Line | End-Line | Branch Target |
| 1 | 0 | 2 | 31 | 31 | 3 |
| 2 | 3 | 4 | 32 | 33 | 1 |
| 3 | 5 | 6 | 34 | 34 | Return |
| Method Name | | | static void main(String arg0[]) | | |
| Method-id | | | 3 | | |
| Block-id | Block-start | Block-end | Start-Line | End-Line | Branch Target |
| 1 | 0 | 10 | 38 | 41 | Return |

Table A.1: Blocks table for Prime example

# Appendix B

# Tool Usage and Screen Shots

In this appendix the usage of the tool is illustrated with an screen shots.

### *Starting GUI:*

GUI can be started by executing the JavaCover.jar file or by executing the command gui or java JavaCover.GUI.GUIDisplay.

### *Instrumenting from GUI:*

A class or package can be instrumented by selecting Instrument tab in the menu bar. Snapshots of instrumentation dialoges are shown in Figure B.3.

```
Instrumentation from command line:
$java JavaCover.Instrument.Instrument
Instrument <instrument types separated by space> <classfile>
-M: Method coverage
-B: Block coverage
-Br: Branch coverage
-P: Preicate coverage
-S: Select coverage
-A: All coverage

eg: $java JavaCover.Instrument.Instrument -A *.class
Time taken to instrument: 4816ms
Number of Classes        : 40
Number of Methods        : 242
Number of Blocks : 1188
Number of Branches       : 1340
```

```
Number of Catches       : 51
Number of Predicates    : 333
Number of Selects       : 3
```

The above example instruments all the classes in the present directory for all coverages.

### *Executing instrumented program:*

There is no change in execution procedure. You can execute the program as normal program.

### *Finding changes made to the code:*

Changes can be found by selecting the change query from database tab. Snapshot of sample change query is shown in Figure B.5

### *Finding test cases:*

Testcases which execute the changed part of the code can be found by selecting test case query from database tab. Snapshot of test case query is shown in Figure B.6.

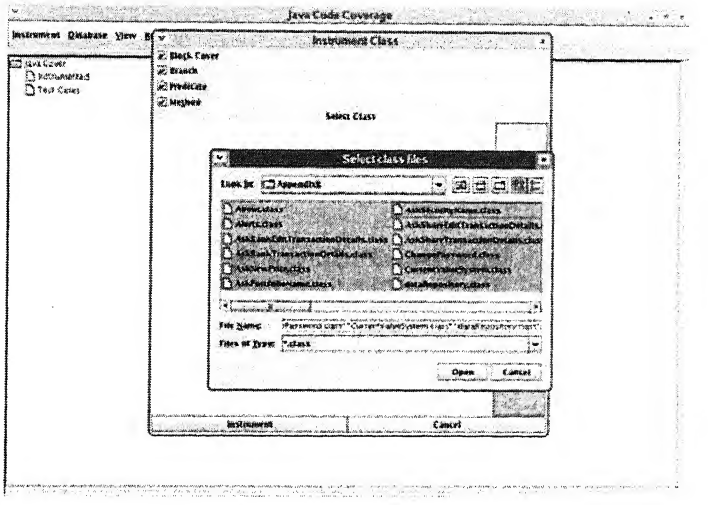



Figure B.1: Instrumentation Dialog

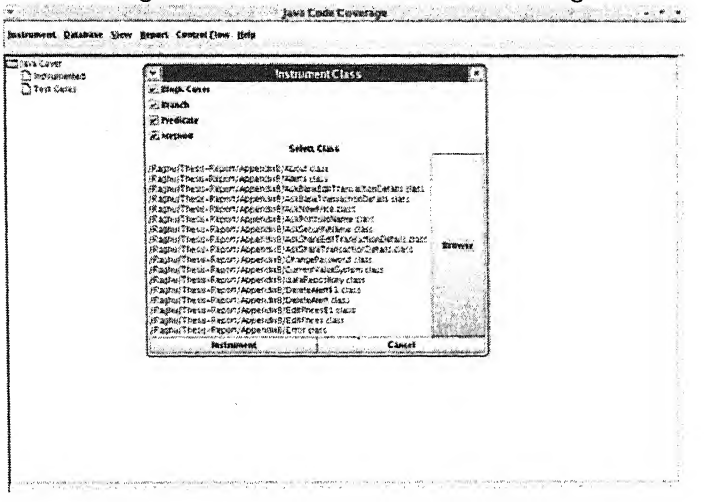



Figure B.2: File chooser for Instrumentation

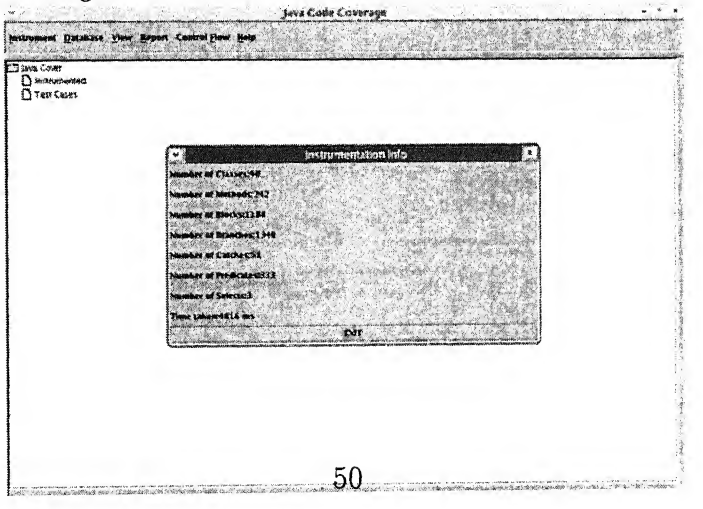





Figure B.3: Instrumentation information

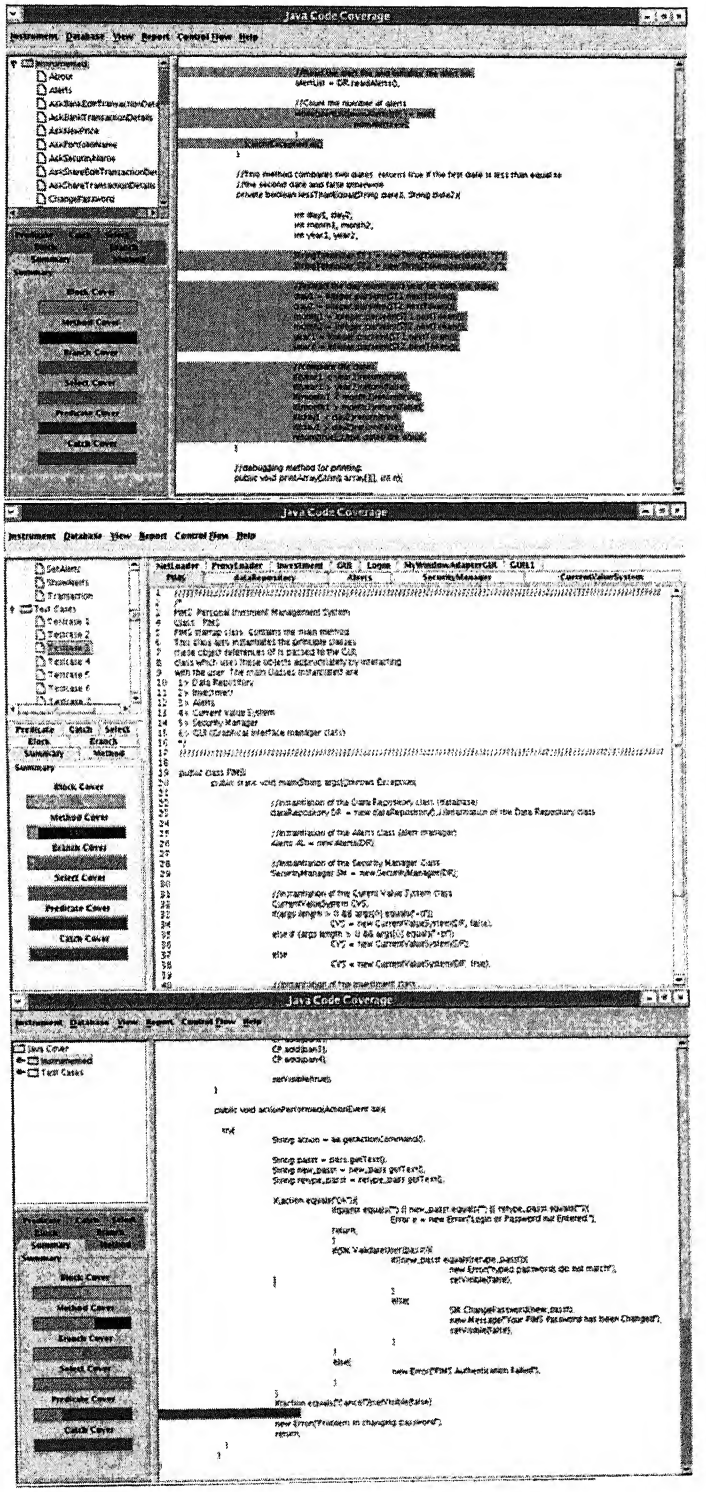



Figure B.4: Coverage view for different coverage measures

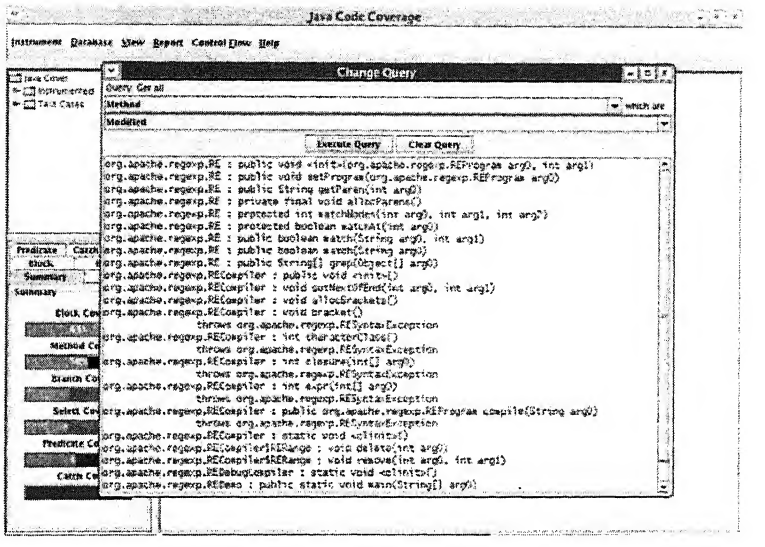



Figure B.5: Finding the chages made to the code

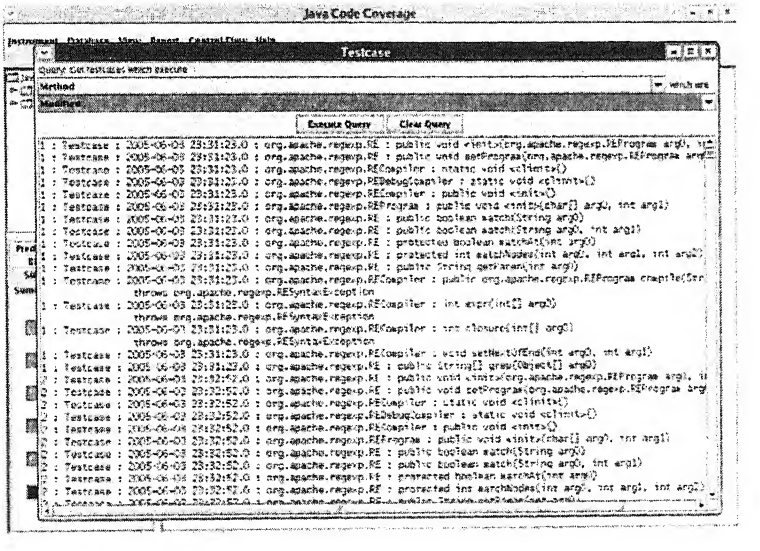



Figure B.6: Finding test cases which execute modified code

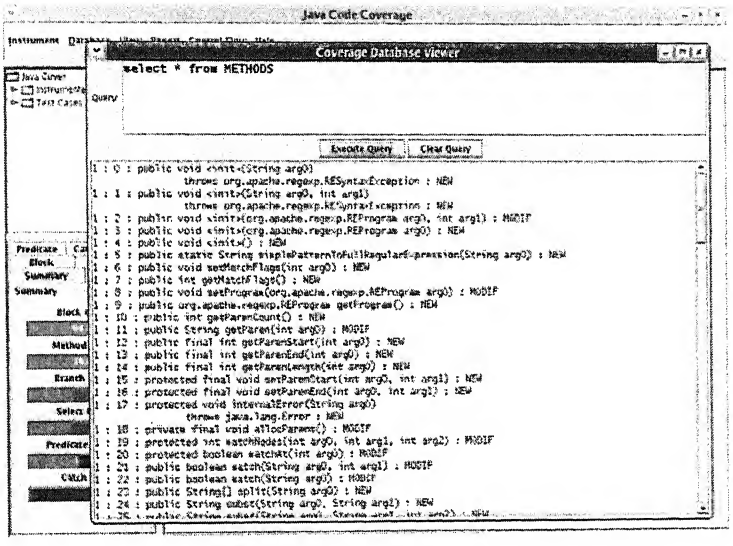



Figure B.7: User defined database query

# References


[1] Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 92, Washington, DC, USA, 2001. IEEE Computer Society.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 2–13, Linz, Austria, september 2004.

[4] Thomas Ball. A theory of predicate-complete test coverage and generation. Technical Report MSR-TR-2004-28, Microsoft Research (MSR), April 2004.

[5] Boris Beizer. *Software Testing Techniques.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[6] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] Bullseye. Homepage: http://www.bullseye.com/coverage.html.

[8] Clover. Clover homepage: http://www.cenqua.com/clover/.

[9] EMMA. Sourceforge homepage: http://emma.sourceforge.net/.

[10] Chengyun Chu Mary Jean Harrold Gregg Rothermel, Roland H. Untch. Test case prioritization: An empirical study, 1999.

[11] Hansel. Sourceforge homepage: http://hansel.sourceforge.net/.

[12] M.J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, November 2001.

[13] Pankaj Jalote. *An integrated approach to software engineering.* Springer-Verlag New York, Inc., 1991.

[14] Junit. Homepage: http://www.junit.org.

[15] Ghulam Mustafa Khan. Test coverage analysis: A method for generic reporting. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, April 2000.

[16] Venkata Sreenivasa Rao N. Test coverage analyzer for java. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, March 2003.

[17] Alessandro Orso, Taweesup Apiwattanapong, James B. Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2004)*, pages 491–500, Edinburgh, Scotland, may 2004.

[18] Atul S. Paldhikar. Coverage based testing and test data generation. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, January 1993.

[19] Quilt. Sourceforge homepage: http://quilt.sourceforge.net/.

[20] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[21] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–106, New York, NY, USA, 2002. ACM Press.

[22] Frank Yellin Tim Lindholm. *The Java(TM) Virtual Machine Specification (2nd Edition).*

[23] Vipindeep V and Pankaj Jalote. Efficient static analysis with path pruning using coverage data. In *ICSE: Workshop on Dynamic Analysis (WODA'05)*, Missouri, USA, May 2005.